Android 上实现水波特效

罗朝辉（http://www.cppblog.com/kesalin）

本文链接：http://www.cppblog.com/kesalin/archive/2010/09/01/android_ripple.html

说明：

本文算法部分整理自 GameRes 上的资料，原作者 Imagic。我只是在学习 Android 的过程中，想到这个特效，然后就在 Android 上实现出来，并在源算法的基础上添加了雨滴滴落特效，以及划过水面时的涟漪特效。 该程序在模拟器和真机上运行速度都较慢，需要进一步优化或使用 JNI 实现，如果你想到好的优化算法，请联系我：kesalin@gmail.com。

示例程序下载：http://www.cppblog.com/Files/kesalin/RippleDemo.zip

基础知识：

在讲解代码之前，我们来回顾一下在高中的物理课上我们所学的关于水波的知识。水波有扩散，衰减，折射，反射，衍射等几个特性：

扩散：当你投一块石头到水中，你会看到一个以石头入水点为圆心所形成的一圈圈的水波，这里，你可能会被这个现象所误导，以为水波上的每一点都是以石头入水点为中心向外扩散的，这是错误的。实际上，水波上的任何一点在任何时候都是以自己为圆心向四周扩散的，之所以会形成一个环状的水波，是因为水波的内部因为扩散的对称而相互抵消了。

衰减：因为水是有阻尼的，否则，当你在水池中投入石头，水波就会永不停止的震荡下去。

折射：因为水波上不同地点的倾斜角度不同，所以我们从观察点垂直往下看到的水底并不是在观察点的正下方，而有一定的偏移。如果不考虑水面上部的光线反射，这就是我们能感觉到水波形状的原因。

反射：水波遇到障碍物会反射。

衍射：在水池中央放上一块礁石，或放一个中间有缝的隔板，那么就能看到水波的衍射现象了。

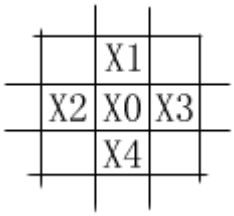
算法推导：

好了，有了这几个特性，再运用数学和几何知识，我们就可以模拟出真实的水波了。但是，如果你曾用 3DMax 做过水波的动画，你就会知道要渲染出一幅真实形状的水波画面少说也得好几十秒，而我们现在需要的是实时的渲染，每秒种至少也得渲染 20 帧才能使得水波得以平滑的显示。考虑到电脑运算的速度，我们不可能按照正弦函数或精确的公式来构造水波，不能用乘除法，更不能用 sin、cos 等三角函数，只能用一种取近似值的快速算法，尽管这种算法存在一定误差，但是为了满足实时动画的要求，我们不得不这样做。

首先我们要建立两个与水池图象一样大小的数组 buf1[PoolWidth * PoolHeight]和 buf2[PoolWidth * PoolHeight]（PoolWidth 为水池图象的象素宽度、PoolHeight 为水池图象的象素高度），用来保存水面上每一个点的前一时刻和后一时刻波幅数据，因为波幅也就代表了波的能量，所以在后面我们称这两个数组为波能缓冲区。水面在初始状态时是一个平面，

各点的波幅都为 0，所以，这两个数组的初始值都等于 0。

下面来推导计算波幅的公式



我们假设存在这样一个一次公式，可以在任意时刻根据某一个点周围前、后、左、右四个点以及该点自身的振幅来推算出下一时刻该点的振幅，那么，我们就有可能用归纳法求出任意时刻这个水面上任意一点的振幅。如左图，你可以看到，某一时刻，X0 点的振幅除了受 X0 点自身振幅的影响外，同时受来自它周围前、后、左、右四个点（X1、X2、X3、X4）的影响（为了简化，我们忽略了其它所有点），而且，这四个点对 X0 点的影响力可以说是机会均等的。那么我们可以假设这个一次公式为：

X0’ = a * (X1 + X2 + X3 + X4) + b * X0　　(公式 1)

a, b 为待定系数，X0’ 为 X0 点下一时刻的振幅，

X0、X1、X2、X3、X4 为当前时刻的振幅

下面我们来求解 a 和 b。

假设水的阻尼为 0。在这种理想条件下，水的总势能将保持不变，水波永远波动。也就是说在任何时刻，所有点的振幅的和保持不变。那么可以得到下面这个公式：

X0’ + X1’ + ... + Xn’ = X0 + X1 + ... + Xn

将每一个点用公式 1 替代，代入上式，得到：

(4a + b) * X0 + (4a + b) * X1 + ... (4a + b) * Xn = X0 + X1 + ... + Xn　=> 4a + b = 1

找出一个最简解：a = 1/2、b = -1。

因为 1/2 可以用移位运算符 “>>” 来进行，不用进行乘除法，所以，这组解是最适用的而且是最快的。那么最后得到的公式就是：

X0’=（X1 + X2 + X3 + X4）/ 2 - X0

好了，有了上面这个近似公式，你就可以推广到下面这个一般结论：已知某一时刻水面上任意一点的波幅，那么，在下一时刻，任意一点的波幅就等于与该点紧邻的前、后、左、右四点的波幅的和除以 2、再减去该点的波幅。

应该注意到，水在实际中是存在阻尼的，否则，用上面这个公式，一旦你在水中增加一个波源，水面将永不停止的震荡下去。所以，还需要对波幅数据进行衰减处理，让每一个点在经过一次计算后，波幅都比理想值按一定的比例降低。这个衰减率经过测试，用 1/32 比

较合适，也就是 1/2^5。可以通过移位运算很快的获得。

到这里，水波特效算法中最艰难的部分已经明了，下面是 Android 源程序中计算波幅数据的代码。

```
// 某点下一时刻的波幅算法为：上下左右四点的波幅和的一半减去当前波幅，即
//     X0' =（X1 + X2 + X3 + X4）/ 2 - X0
//  +----x3----+
//  +     |     +
//  +     |     +
// x1---x0----x2
//  +     |     +
//  +     |     +
//  +----x4----+
//
void rippleSpread()
{
    int pixels = m_width * (m_height - 1);
    for (int i = m_width; i < pixels; ++i) {
        // 波能扩散:上下左右四点的波幅和的一半减去当前波幅
        // X0' =（X1 + X2 + X3 + X4）/ 2 - X0
        //
        m_buf2[i] =
            (short)(((m_buf1[i - 1] + m_buf1[i + 1]+
            m_buf1[i - m_width] + m_buf1[i + m_width]) >> 1)
             - m_buf2[i]);

        // 波能衰减 1/32
        //
        m_buf2[i] -= m_buf2[i] >> 5;
    }

    //交换波能数据缓冲区
    short[] temp = m_buf1;
    m_buf1 = m_buf2;
    m_buf2 = temp;
}
```

渲染：

然后我们可以根据算出的波幅数据对页面进行渲染。

因为水的折射，当水面不与我们的视线相垂直的时候，我们所看到的水下的景物并不是在观察点的正下方，而存在一定的偏移。偏移的程度与水波的斜率，水的折射率和水的深度都有关系，如果要进行精确的计算的话，显然是很不现实的。同样，我们只需要做线性的近似处理就行了。因为水面越倾斜，所看到的水下景物偏移量就越大，所以，我们可以近似的用水面上某点的前后、左右两点的波幅之差来代表所看到水底景物的偏移量。

在程序中，用一个页面装载原始的图像，用另外一个页面来进行渲染。先取得指向两个页面内存区的指针 src 和 dst，然后用根据偏移量将原始图像上的每一个象素复制到渲染页面上。进行页面渲染的代码如下：

```
void rippleRender()
{
    int offset;
    int i = m_width;
    int length = m_width * m_height;
    for (int y = 1; y < m_height - 1; ++y) {
       for (int x = 0; x < m_width; ++x, ++i) {
          // 计算出偏移象素和原始象素的内存地址偏移量 :
          //offset = width * yoffset + xoffset
          offset = (m_width * (m_buf1[i - m_width] - m_buf1[i +
m_width])) + (m_buf1[i - 1] - m_buf1[i + 1]);

          // 判断坐标是否在范围内
          if (i + offset > 0 && i + offset < length) {
             m_bitmap2[i] = m_bitmap1[i + offset];
          }
          else {
             m_bitmap2[i] = m_bitmap1[i];
          }
       }
    }
}
```

增加波源：

俗话说：无风不起浪，为了形成水波，我们必须在水池中加入波源，你可以想象成向水中投入石头，形成的波源的大小和能量与石头的半径和你扔石头的力量都有关系。知道了这些，那么好，我们只要修改波能数据缓冲区 buf，让它在石头入水的地点来一个负的“尖脉冲”，即让 buf[x,y] = -n。经过实验，n 的范围在（32 ~ 128）之间比较合适。

控制波源半径也好办，你只要以石头入水中心点为圆心，画一个以石头半径为半径的圆，让这个圆中所有的点都来这么一个负的“尖脉冲”就可以了（这里也做了近似处理）。
增加波源的代码如下：

```
// stoneSize    : 波源半径
// stoneWeight  : 波源能量
//
void dropStone(int x, int y, int stoneSize, int stoneWeight)
{
    // 判断坐标是否在范围内
    if ((x + stoneSize) > m_width || (y + stoneSize) > m_height
            || (x - stoneSize) < 0 || (y - stoneSize) < 0) {
        return;
    }

    int value = stoneSize * stoneSize;
    short weight = (short)-stoneWeight;
    for (int posx = x - stoneSize; posx < x + stoneSize; ++posx)
    {
        for (int posy = y - stoneSize; posy < y + stoneSize; ++posy)
        {
            if ((posx - x) * (posx - x) + (posy - y) * (posy - y)
                < value)
            {
                m_buf1[m_width * posy + posx] = weight;
            }
        }
    }
}
```

如果我们想要模拟在水面划过时引起的涟漪效果，那么我们还需要增加新的算法函数 breasenhamDrop。

```
void dropStoneLine(int x, int y, int stoneSize, int stoneWeight)
{
    // 判断坐标是否在屏幕范围内
    if ((x + stoneSize) > m_width || (y + stoneSize) > m_height
        || (x - stoneSize) < 0 || (y - stoneSize) < 0) {
            return;
    }

    for (int posx = x - stoneSize; posx < x + stoneSize; ++posx)
    {
        for (int posy = y - stoneSize; posy < y + stoneSize; ++posy)
        {
            m_buf1[m_width * posy + posx] = -40;
        }
    }
}

// xs, ys : 起始点，xe, ye : 终止点
// size : 波源半径，weight : 波源能量
void breasenhamDrop (int xs, int ys, int xe, int ye, int size, int
weight)
{
    int dx = xe - xs;
    int dy = ye - ys;
    dx = (dx >= 0) ? dx : -dx;
    dy = (dy >= 0) ? dy : -dy;

    if (dx == 0 && dy == 0) {
        dropStoneLine(xs, ys, size, weight);
    }
    else if (dx == 0) {
        int yinc = (ye - ys != 0) ? 1 : -1;
        for(int i = 0; i < dy; ++i){
            dropStoneLine(xs, ys, size, weight);
            ys += yinc;
        }
    }
    else if (dy == 0) {
        int xinc = (xe - xs != 0) ? 1 : -1;
        for(int i = 0; i < dx; ++i){
```

```
            dropStoneLine(xs, ys, size, weight);
            xs += xinc;
        }
    }
    else if (dx > dy) {
        int p = (dy << 1) - dx;
        int inc1 = (dy << 1);
        int inc2 = ((dy - dx) << 1);
        int xinc = (xe - xs != 0) ? 1 : -1;
        int yinc = (ye - ys != 0) ? 1 : -1;

        for(int i = 0; i < dx; ++i) {
            dropStoneLine(xs, ys, size, weight);
            xs += xinc;
            if (p < 0) {
                p += inc1;
            }
            else {
                ys += yinc;
                p += inc2;
            }
        }
    }
    else {
        int p = (dx << 1) - dy;
        int inc1 = (dx << 1);
        int inc2 = ((dx - dy) << 1);
        int xinc = (xe - xs != 0) ? 1 : -1;
        int yinc = (ye - ys != 0) ? 1 : -1;

        for(int i = 0; i < dy; ++i) {
            dropStoneLine(xs, ys, size, weight);
            ys += yinc;
            if (p < 0) {
                p += inc1;
            }
            else {
                xs += xinc;
                p += inc2;
            }
        }
    }
}
```

效果图：

划过水面时的涟漪特效

雨滴滴落水面特效

结语：

这种用数据缓冲区对图像进行水波处理的方法，有个最大的好处就是，程序运算和显示的速度与水波的复杂程度是没有关系的，无论水面是风平浪静还是波涛汹涌，程序的 fps 始终保持不变，这一点你研究一下程序就应该可以看出来。

罗朝辉

2010-09-01

补记：

关于优化的问题，上面的算法是针对每一个像素进行波幅计算，效率比较低，尤其是在手机上运行，相当缓慢。我们可以利用线性插值进行优化，这样可以将计算减少一半（MeshSize 为 2）或减少四分之三（MeshSize 为 4）。

在下面的代码中，为了充分使用移位运算替代乘除法，MeshSize 必须为 2 的整次幂，MeshShift 就是其幂数，表示计算时的移位位数。

线性插值优化之后的水波扩散代码如下：

```
static final int MeshSize = 2;
static final int MeshShift = 1;
int m_meshWidth;
int m_meshHeight;
```

```
m_meshWidth = m_width / MeshSize + 1;
m_meshHeight = m_height / MeshSize + 1;;

void rippleSpread()
{
       m_waveFlag = false;

       int i = 0, offset = 0;
       m_buf2[0] = (short) (((m_buf1[1] + m_buf1[m_meshWidth]) >>
1) - m_buf2[0]);

       // first column
       for (int y = 1; y < m_meshHeight - 1; ++y) {
          i += m_meshWidth;
          m_buf2[i] = (short) (((m_buf1[i + 1] + m_buf1[i -
m_meshWidth]
                + m_buf1[i + m_meshWidth]) >> 1) - m_buf2[i]);

          m_buf2[i] -= (m_buf2[i] >> 5);
          m_waveFlag |= (m_buf2[i] != 0);
       }

       // first row
       for (i = 1; i < m_meshWidth - 1; ++i) {
          m_buf2[i] = (short) (((m_buf1[i - 1] + m_buf1[i + 1]
                + m_buf1[i + m_meshWidth]) >> 1) - m_buf2[i]);

          m_buf2[i] -= (m_buf2[i] >> 5);
          m_waveFlag |= (m_buf2[i] != 0);
       }

       for (int y = 1; y < m_meshHeight - 1; ++y) {
          offset += m_meshWidth;
          for (int x = 1; x < m_meshWidth - 1; ++x) {
             i = offset + x;

             // 波能扩散:上下左右四点的波幅和的一半减去当前波幅
             // X0' =（X1 + X2 + X3 + X4）/ 2 - X0
             //
             m_buf2[i] = (short) (((m_buf1[i - 1] + m_buf1[i + 1]
                    + m_buf1[i - m_meshWidth]
                    + m_buf1[i + m_meshWidth]) >> 1)
                    - m_buf2[i]);
```

```
                // 波能衰减 1/32
                //
                m_buf2[i] -= (m_buf2[i] >> 5);

                m_waveFlag |= (m_buf2[i] != 0);
            }
        }

        //if (m_waveFlag)
        {
            m_waveFlag = false;
            offset = 0;
            for (int y = 1; y < m_meshHeight - 1; ++y) {
                offset += m_meshWidth;
                for (int x = 1; x < m_meshWidth - 1; ++x) {
                    i = offset + x;
                    m_bufDiffX[i] = (short) ((m_buf2[i + 1] - m_buf2[i
- 1]) >> 3);
                    m_bufDiffY[i] = (short) ((m_buf2[i + m_meshWidth]
- m_buf2[i - m_meshWidth]) >> 3);

                    m_waveFlag |= (m_bufDiffX[i] != 0 ||
m_bufDiffY[i] != 0);
                }
            }
        }

        //交换波能数据缓冲区
        short[] temp = m_buf1;
        m_buf1 = m_buf2;
        m_buf2 = temp;
}
```

既然波幅计算使用的是线性插值，描绘时的代码也许相应进行更改：

```
Point p1, p2, p3, p4;
Point pRowStart, pRowEnd, p, rowStartInc, rowEndInc, pInc;

void rippleRender()
{
    int px = 0, py = 0, dx = 0, dy = 0;
    int index = 0, offset = 0, pyOffset = 0;

    for (int j = 1; j < m_meshHeight; ++j) {
```

```
        offset += m_meshWidth;
        for (int i = 1; i < m_meshWidth; ++i) {
            index = offset + i;
            p1.x = m_bufDiffX[index - m_meshWidth - 1];
            p1.y = m_bufDiffY[index - m_meshWidth - 1];
            p2.x = m_bufDiffX[index - m_meshWidth];
            p2.y = m_bufDiffY[index - m_meshWidth];
            p3.x = m_bufDiffX[index - 1];
            p3.y = m_bufDiffY[index - 1];
            p4.x = m_bufDiffX[index];
            p4.y = m_bufDiffY[index];

            pRowStart.x = p1.x << MeshShift;
            pRowStart.y = p1.y << MeshShift;
            rowStartInc.x = p3.x - p1.x;
            rowStartInc.y = p3.y - p1.y;

            pRowEnd.x = p2.x << MeshShift;
            pRowEnd.y = p2.y << MeshShift;
            rowEndInc.x = p4.x - p2.x;
            rowEndInc.y = p4.y - p2.y;

            py = (j - 1) << MeshShift;
            for (int y = 0; y < MeshSize; ++y) {
                p.x = pRowStart.x;
                p.y = pRowStart.y;

                // scaled by MeshSize times
                pInc.x = (pRowEnd.x - pRowStart.x) >> MeshShift;
                pInc.y = (pRowEnd.y - pRowStart.y) >> MeshShift;

                px = (i - 1) << MeshShift;
                pyOffset = py * m_width;
                for (int x = 0; x < MeshSize; ++x) {
                    dx = px + p.x >> MeshShift;
                    dy = py + p.y >> MeshShift;

                    if ((dx >= 0) && (dy >= 0) && (dx < m_width) && (dy
< m_height) ) {
                        m_bitmap2[pyOffset + px] = m_bitmap1[dy *
m_width + dx];
                    }
                    else {
                        m_bitmap2[pyOffset + px] = m_bitmap1[pyOffset
```

```
+ px];
                }

                p.x += pInc.x;
                p.y += pInc.y;
                ++px;
            }

            pRowStart.x += rowStartInc.x;
            pRowStart.y += rowStartInc.y;
            pRowEnd.x += rowEndInc.x;
            pRowEnd.y += rowEndInc.y;
            ++py;
        }
    }
}
```